Universiteit Twente
Afdeling Informatica
4 januari 2011

Kwartiel 1011/2
2^e huiswerkserie

Algoritmen, Datastructuren en Complexiteit (214020)

Uitwerkingen

De **deadline** voor het inleveren van deze huiswerkserie is maandagochtend 17 januari 2011, om 9.30 uur. U levert het werk in in het postvakje van uw werkcollegedocent. Bij de opgaven waar om een algoritme wordt gevraagd, geeft u de pseudo-code van uw oplossing en een beknopte maar duidelijke uitleg van de werking. Plagiaat zal streng worden bestraft.

Deze opgave zal worden beoordeeld met twee minnen, een min, een voldoende, of een plus. Deze huiswerkserie kan maximaal beloond worden met 0.3 punt extra bij het tentamencijfer, onder de voorwaarden die bij de spelregels worden uitgelegd. Veel succes!

1. Gegeven een ongerichte graaf, gerepresenteerd als de adjacency list van een gerichte symmetrische graaf. Geef een DFS algoritme die bepaalt of deze graaf een cykel bevat (hint: geef aan een DFS aanroep de voorganger p van een vertex v mee; een edge vp betekent namelijk niet dat er een cykel in de ongerichte graaf is!).

   Uitwerking:

   We passen het standaard DFS skelet (sheets 26 en 27 van hoorcollege 5) aan door een boolean *cyclefound* waar te maken op het moment dat we ergens in de DFS een edge vinden naar een gele (dus reeds eerder in het pad bezochte) vertex, behalve als dit een edge is naar een directe voorganger. Vandaar dat we de directe voorganger meegeven aan *cycleDFS*; bij de eerste aanroep is er geen directe voorganger en geven we een dummywaarde -1 mee. De DFS wordt gestopt op het moment dat een cykel gevonden is.

```
bool detectcycle(intlist[] adjV, int n)
{ int[1..n] color;
  bool cyclefound = false;
  for (int v=1; v<=n; v++) color[v] = blue;
  while v<=n and not cyclefound
  { if color[v]==blue
```

```
        cycleDFS(adjV, color, v, -1)
  }
  return cyclefound
}

void cycleDFS(intlist[] ajdV, int[] color, int v, int p)
{ int w; intlist remAdj = adjV[v];
  color[v] = yellow;
  while not remAdj.isEmpty() and not cyclefound
  {w = remAdj.first();
   remAdj = remAdj.rest();
   if color[w]==blue
      then cycleDFS(adjV, color, w, v)
      else if color[w]== yellow and w!=p
              then cyclefound = true
  }
  color[v] = green
}
```

2. Gegeven een gerichte graaf, gerepresenteerd als een adjacency matrix. Geef een algoritme die bepaalt of deze graaf een boom is, dwz. hij bevat geen cykels, en er is een knoop R zodat alle andere knopen via precies 1 pad vanuit R te bereiken zijn (hint: pas Warshall aan zodat bijgehouden wordt *hoeveel* paden er tussen knoop i en j zijn).

   Uitwerking:

   Pas Warshall aan als volgt:

   ```
   R[i,j] = R[i,j] + R[i,k]*R[k,j]
   ```

   Verder moet ook het stuk eruit gehaald worden waarbij de waarden op de diagonaal TRUE gemaakt worden (de reflexive closure!).

   Nu zal $R[i, j]$ uiteindelijk het aantal paden tussen i en j bevatten.

   Het checken gaat nu als volgt:

   - Check of alle waarden op de hoofddiagonaal 0 zijn: dit betekent dat de graaf cykelvrij is.
   - Check of alle andere waarden hooguit 1 zijn; dit betekent dat alle paden uniek zijn.
   - Check of er een vertex j is, met alle kolomwaarden voor j een 0, en alle rijwaarden (behalve $R[j, j]$ een 1. Dit kan zoals in het werkcollege behandeld bij opgave 5.7.

3. Een 4-clique is een deelgraaf met 4 vertices die compleet is, dus edges tussen elke twee vertices. Gegeven een ongerichte graaf in adjacency matrix representatie met n vertices. Geef een algoritme dat in polynomiale tijd bepaalt of deze graaf een 4-clique heeft. Hint: wat is het aantal deelverzamelingen van $1..n$ met 4 elementen?

   Uitwerking:

   Stel er zijn n vertices; het aantal deelverzamelingen met 4 vertices is $n(n-1)(n-2)(n-3)/24$. Per deelverzameling moeten we de aanwezigheid van 6 edges checken. Dus als we simpelweg alle deelverzamelingen afgaan, hebben we een algoritme met complexiteit $\Theta(n^4)$, dus polynomiaal. Het algoritme:

```
bool 4clique(bool[][]A, int n)
{ for (int i=1; i<=n; i++)
    for  (int j=i+1; j<=n; j++)
      for  (int k=j+1; k<=n; k++)
        for  (int l=k+1; l<=n; l++)
            if (A[i,j]&&A[j,k]&&A[k,l]&&
                A[i,k]&&A[i,l]&&A[j,l]) return true;
  return false;
}
```

4. Gegeven een array E met lengte n, dat n verschillende integers bevat.

   (a) Geef een algoritme dat de lengte van de langste stijgende subreeks van E vindt. Zo'n subreeks hoeft niet opeenvolgende te zijn: bijvoorbeeld, de langste stijgende subreeks van 11,17,5,8,6,4,7,12,3 is 5,6,7,12. Hint: laat $A[i]$ de lengte zijn van de langste stijgende subreeks die begint met $E[i]$, geef een recursieve uitdrukking voor $A[i]$, en bepaal de lengte van de langste stijgende subreeks aan de hand van A.

   Uitwerking:

   Analyse: de lengte van de langste subreeks die met $E[i]$ begint, is 1 plus de lengte van de langste subreeks na i waar je $E[i]$ voor kan plakken. Dus:
   $A[i] = 1 + \max_{i<j\leqslant n}\{A[j] \mid E[i] < E[j]\}$
   en de gevraagde lengte: $lengte = \max_{1\leqslant i\leqslant n}\{A[i]\}$
   Het algoritme:

```
int lls(int[] E, int n)
{
  int[1..n] A;
```

```
  int maxl;

  for (i=n; i>=1; i--)
   { maxl=0;
     for (j=i+1; j<=n; j++)
       if E[i]<E[j] then maxl = max(maxl, A[j])

     A[i] = maxl +1
   }

  maxl=0;
  for (i=1; i<=n; i++)
   maxl = max(maxl, A[i])
  return maxl
}
```

(b) Zorg dat de algoritme ook bepaalt wat de langste stijgende subreeks is.

Uitwerking:

We nemen aan dat er globaal een array $int[0, n]loc$ gedefinieerd is. Uiteindelijk zal $loc[0]$ de index bevatten van het eerste element in de langste subreeks, $loc[loc[0]]$ de index van het tweede element, enzovoorts; als $loc[k] == 0$ is k de index van het laatste element van de subreeks.

```
int lls(int[] E, int n)
{
  int[1..n] A;
  int maxl;

  for (i=n; i>=1; i--)
   { maxl=0;
     loc[i]=0;
     for (j=i+1; j<=n; j++)
       if E[i]<E[j] and A[j]>maxl then {maxl=A[j]; loc[i]=j}

     A[i] = maxl+1
   }

  maxl=0;
  for (i=1; i<=n; i++)
   if A[i]>maxl then {maxl=A[i]; loc[0]=i}
  return maxl
}
```

5. Beargumenteer voor de volgende uitspraken of ze waar of niet waar zijn:

(a) Het handelsreizigerprobleem is reduceerbaar tot het satisfiability probleem (SAT).

Uitwerking:

Het handelsreizigersprobleem zit in NP, SAT zit in NPC, dus de bewering is waar.

(b) Stel P $\neq$ NP, en we hebben een probleem $X \notin$ P, en alle problemen in NP zijn reduceerbaar tot X. Dan geldt $X \in$ NP.

Uitwerking:

De bewering is niet waar: X is NP-hard, maar we weten niet of X in NP zit.

(c) Iedere ongerichte graaf G is in polynomiale tijd te transformeren naar een graaf G' zodat tussen G en G' de volgende relatie bestaat: Als G' met $n+1$ kleuren te kleueren is, dan is G met n kleuren te kleuren (zonder buren met dezelfde kleur).

Uitwerking:

Is waar. Construeer G' uit $G = \langle V, E \rangle$ door aan G een knoop $w \notin V$ toe te voegen, en door E uit te breiden met kanten die iedere knoop $v \in V$ met de toegevoegde knoop w verbinden (d.w.z. met de verzameling $\{vw | v \in V\}$. Een kleuring van deze G' met $n+1$ kleuren bevat een kleuring van G met n kleuren. Immers: alle knopen van de oorspronkelijke G hebben een andere kleur dan de kleur van de toegevoegde w.

(d) Het n-kleur probleem voor grafen is polynomiaal reduceerbaar tot het $n+1$-kleur probleem.

Uitwerking:

Is waar, volgt uit de vorige bewering.